GAGE: Genetic Algorithm-based Graph Explainer for Malware Analysis

Mohd Saqib Benjamin C. M. Fung
School of Information Studies
McGill University, Canada
mohd.saqib@mail.mcgill.ca & ben.fung@mcgill.ca

Philippe Charland
Mission Critical Cyber Security Section,
Defence R&D Canada
philippe.charland@drdc-rddc.gc.ca

Andrew Walenstein
BlackBerry Limited
Waterloo, Canada
walenste@ieee.org

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 CRWN | DOI: 10.1109/ICDE60146.2024.00179

***Abstract*—Malware analysts often prefer reverse engineering using Call Graphs, Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs), which involves the utilization of black-box Deep Learning (DL) models. The proposed research introduces a structured pipeline for reverse engineering-based analysis, offering promising results compared to state-of-the-art methods and providing high-level interpretability for malicious code blocks in subgraphs. We propose the Canonical Executable Graph (CEG) as a new representation of Portable Executable (PE) files, uniquely incorporating syntactical and semantic information into its node embeddings. At the same time, edge features capture structural aspects of PE files. This is the first work to present a PE file representation encompassing syntactical, semantic, and structural characteristics, whereas previous efforts typically focused solely on syntactic or structural properties. Furthermore, recognizing the limitations of existing graph explanation methods within Explainable Artificial Intelligence (XAI) for malware analysis, primarily due to the specificity of malicious files, we introduce Genetic Algorithm-based Graph Explainer (GAGE). GAGE operates on the CEG, striving to identify a precise subgraph relevant to predicted malware families. Through experiments and comparisons, our proposed pipeline exhibits substantial improvements in model robustness scores and discriminative power compared to the previous benchmarks. Furthermore, we have successfully used GAGE in practical applications on real-world data, producing meaningful insights and interpretability. This research offers a robust solution to enhance cybersecurity by delivering a transparent and accurate understanding of malware behaviour. Moreover, the proposed algorithm is specialized in handling graph-based data, effectively dissecting complex content and isolating influential nodes.**

***Index Terms*—malware analysis, explainable AI, interpretability, graph, genetic algorithm**

I. INTRODUCTION

Malware poses an ever-growing threat in the digital landscape, with reports from VirusTotal[1] indicating a 27% increase in computer viruses in 2021 alone. Concurrently, a study by Kaspersky[2] highlights the detection of approximately 5.2% of the 360,000 new malicious files each day. Traditional malware analysis methods are struggling to cope with the influx of this expanding and increasingly obfuscated malware [1]. In response to these challenges, this research proposed a graph-based representation for executable files [2], aiming to enhance the precision and accuracy of malware behaviour identification with a robust explanation. The motivation behind this work lies in the potential of graph representations to capture the semantic, syntactical, and flow control aspects of programs and data, thus enabling the more accurate detection of malicious behaviour. To achieve this, we employ DL techniques for graph classification. However, DL models often suffer from the "black-box" drawback, necessitating the development of a state-of-the-art graph classification explanation algorithm tailored specifically for malware analysis.

Malware analysis mainly encompasses two approaches: static [3] and dynamic analysis [4]. Static analysis involves extracting features from PE files, such as numerical attributes, printable strings, and import and export information [5]. Dynamic analysis, on the other hand, observes the behaviour of a malicious file in a controlled environment, analyzing dynamic features like registry changes, memory utilization, and network activity [4]. While hybrid analysis combines both static and dynamic features, these manual-intensive methods face limitations, including their inefficiency in handling the increasing number of malware samples, struggles in distinguishing between various malware families, and susceptibility to zero-day exploits, obfuscation, or polymorphic malware. As a result, researchers have shifted towards ML-based analysis [6], which has its own challenges, including intensive manual feature engineering [7] and the incorporation of various data types, such as images and assembly code. Moreover, DL methods have been employed, but interpretability and human understanding of model outputs are needed [8].

Despite significant work in automated malware detection, only few studies have thoroughly explored the potential of graph representations for executables. The CFG [9] is a well-known representation, followed by the DFG [10]. However, these representations do not capture the critical function call properties within nodes or blocks. To address this limitation, we propose a new executable representation, the CEG, that retains semantic information by processing code blocks using an Attention-based Autoencoder (AED). AED learns the order of instructions and generates embeddings accordingly. Additionally, we extract statistical distributions of opcodes and operands, combining them with AED-generated features to

This research is supported by BlackBerry Limited (ALLRP 561035), Defence Research & Development Canada, and NSERC Alliance Grants (ALLRP 561035-20)

[1]https://assets.virustotal.com/reports/2021trends.pdf

[2]https://www.kaspersky.com/about/press-releases/2020 the-number-of-new-malicious-files-detected every-day-increases-by-52-to-360000-in-2020

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00179

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.

incorporate syntactical information. Following the generation and classification of CEG, a significant challenge arises in providing interpretable reasoning behind the classification—a crucial aspect for human understanding and practical malware analysis, even without prior knowledge of the model. To address this challenge, we employ GAGE, an explanation extraction method for subgraphs from CEGs, which offers insights into malicious intent code blocks.

The major contributions of this research include:

- We have introduced a novel representation for PE files, CEG, that incorporates both syntactical and semantical details into the embedding of its nodes. Notably, CEG is the first executable representation to feature edge attributes that capture the control flow of both external and intra-function calls. To generate embeddings for the nodes within CEG, we introduced the AED. This model has been trained on a dataset comprising one million code blocks, enabling it to efficiently generate block-level encodings for assembly code. These embeddings effectively capture fine-grained information on instruction order, semantics, and code intent. In robustness tests, features generated by AED have demonstrated superior performance compared to state-of-the-art methods for detecting malware behaviour.
- We introduce GAGE, a specialized model-agnostic graph explainer designed explicitly for the intricate task of malware behaviour detection. Operating on CEGs, GAGE employs a genetic algorithm to iteratively refine subgraphs with the primary aim of minimizing the Euclidean distance between the original graph's softmax probability distribution and that of the extracted subgraph. This optimization process enables GAGE to uncover precise subgraphs that capture vital aspects of malware behaviour, addressing the limitations faced by previous graph explainers in malware analysis. Prior methods, such as gradient-based, surrogate, decomposition-based, and model-level explainers, struggled to provide effective explanations for graph-based malware analysis, due to the complexities of malware files, which often encompass mixed content and intricate relationships within graphs. In contrast, GAGE's innovative approach represents a significant advancement in malware analysis and graph-based model explainability, successfully bridging the gap between complex malware behaviour and interpretable model outputs.
- Through experiments and comparisons with state-of-the-art methods, our proposed pipeline achieves a 31% improvement in robustness score compared to the previous benchmarks. This significant enhancement in robustness demonstrates the effectiveness of our approach in distinguishing between different malware families. Additionally, our approach results in a substantial improvement in discriminative power, with a 9% increase in precision, a 7% increase in recall, and a 4% increase in accuracy. These improvements are pivotal for precise malware detection and classification in the ever-evolving threat landscape, making our model a valuable asset in increasing cybersecurity measures.
- Application of the proposed algorithm to malware data, showcasing GAGE's novelty in graph-based learning. It is particularly effective for challenging graph data where identifying influential nodes is difficult. GAGE serves as an XAI method, overcoming limitations associated with other algorithms in handling graph data.

Generally, the innovative solution provided by GAGE can significantly impact graph-based learning and explanation approaches by providing a novel model-agnostic graph explainer that addresses the complexities associated with graph data. GAGE's utilization of a genetic algorithm to iteratively refine subgraphs enhances its ability to capture intricate relationships within graphs, making it applicable to various domains beyond malware detection. By focusing on extracting precise subgraphs, GAGE enhances the interpretability of graph-based models, facilitating a deeper understanding of complex data structures and their underlying patterns. This advancement can lead to improved decision-making in fields such as social network analysis, bioinformatics, and financial risk assessment, where understanding the relationships and interactions within large-scale graphs is crucial. Additionally, GAGE's approach may inspire further developments in XAI, promoting transparency and trustworthiness in graph-based machine learning models across diverse applications.

The paper organisation is as follows: In the second section, we discuss the background of malware analysis using graphs and the challenges in explaining graph-based classification. The third section covers the problem formulation, while the fourth section details the model development process. The fifth section presents the results and discussion, followed by the conclusion.

II. Background

A. Graph in malware analysis

Malware analysis has increasingly incorporated graph-based approaches, as they provide valuable insights into function call flows and recurring patterns within executables. Various types of graphs are now employed by researchers for analysis using Machine Learning (ML) and DL techniques. For instance, CFG models control flow relationships among code's basic blocks, enabling the detection of malicious behaviour by capturing execution paths and control transfers within malicious files [11]–[13]. Yan et al. [11] utilized Graph Convolutional Neural Networks (GCNN) to embed structural information from CFGs, facilitating malware classification. In a different approach, Nguyen et al. [12] converted CFGs into images and performed image classification for faster and cost-effective analysis compared to direct CFG analysis. Additionally, they extracted statistical opcode features, created node features in CFG, and conducted classification using GNNs, along with providing explanations for the classification process [12].

Another commonly used graph is the Call Graph, which illustrates calling relationships between program functions or

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.

methods. It reveals how functions invoke each other, offering insights into the execution flow and dependencies within malicious files [14]. Kinable et al. [14] extracted features and performed graph similarity-based analysis to detect similar patterns in malware. Nevertheless, this approach shares similarities with signature-based malware detection and can be evaded. Likewise, Hassesn et al. [15] presented a scalable method for malware detection based on call graph features. However, their features lack dynamicity.

On the other hand, DFGs track data and variable flows within a program, aiding in the analysis of data manipulation, transformation, and sharing across code segments. This enables the automated identification of potentially malicious data operations or information leakage [16], [17]. Wuchner et al. [16] conducted quantitative heuristic analysis on the data flow of executables, achieving effective malware detection. Similarly, they performed nearly identical analysis using n-gram analysis on DFGs [17].

While the algorithms mentioned above have achieved commendable classification and detection levels, they suffer from two significant issues. Firstly, they need to consider the semantic understanding of code analysis within these graphs, a necessity for understanding executable functionality amidst obfuscation and polymorphism. Secondly, they are black-box algorithms that lack enhanced interpretability for malware analysts or cybersecurity stakeholders. In contrast, Herath et al. [13] discussed explainability and presented it as subgraphs of CFGs. Nevertheless, their approach relied on statistical opcode stratification, susceptible to manipulation through obfuscation or adversarial attacks. Thus, their extracted subgraphs, while offering explainability, may not be as robust as required for diverse malware families and benign samples.

B. Challenges in explainability methods

The field of explainability in graph-based models presents several challenges, due to the unique characteristics of graphs and the fact that existing methods have limitations when applied to malware analysis tasks. In this section, we discuss various types of explainability algorithms and why they may not be suitable for effectively explaining malware behaviour, highlighting the need for our proposed GAGE framework.

Methods such as Sensitive Analysis (SA) [18], Guided Backpropagation (GBP) [18], Class Activation Mapping (CAM) [19], and Gradient-weighted CAM (Grad-CAM) [19] are popular gradient and perturbation-based approaches. However, these methods may face challenges when dealing with malicious files that contain both benign and malicious code. In such cases, these algorithms could be misled by the benign code, resulting in incomplete or incorrect explanations. Additionally, when the graph comprises benign and malicious nodes and edges, these methods may assign equal importance to both types, leading to diluted explanations that fail to identify key malicious behaviours.

Surrogate methods, such as GraphLIME [20], Relational model explainer (RelEx) [21], and Probabilistic Graphical Model explanations (PGM-Explainer) [22], rely on linear classification models, which may not effectively capture the complex and non-linear behaviour of malicious files. These files often exhibit mixed behaviour, making it challenging for traditional linear models to distinguish between benign and malicious nodes and edges accurately. Moreover, building surrogate models, such as GraphLIME [20], typically involves creating many training samples using perturbation techniques. However, applying perturbations to malicious code may not reflect real-world scenarios or provide meaningful insights, limiting the effectiveness of these surrogate models.

Excitation Backpropagation (EB) [19], GNN-layer-wise Relevance Propagation (LRP) [23], and decomposition-based algorithms may not be suitable for explaining malicious files. These methods often decompose the graph randomly or mask nodes, without considering their actual relevance to the file's behaviour.

Model-level explanations, such as the XGNN approach [24], may not capture the intricate interactions between different nodes and edges crucial for understanding malicious behaviour at the local or file level. XGNN, based on reinforcement learning, requires the selection of a starting node to generate explanations or subgraphs. This approach may overlook isolated nodes or graphs not directly connected to the selected node, limiting its ability to provide comprehensive insights into malware behaviour.

Existing explainability methods face challenges in effectively elucidating malicious behaviour in graph-based malware analysis. These challenges arise, due to the complex nature of malware files, which often contain mixed content and intricate relationships between graph elements. Therefore, our proposed GAGE framework aims to address these limitations and provide robust explanations tailored to the unique characteristics of malware graphs.

III. Problem Formulation

In this section, we formulate three critical problems addressed in this research, all centred around the analysis and interpretation of disassembled binary code.

A. Problem 1: CEG construction

The first problem revolves around constructing a CEG from disassembled binary code. We aim to represent the code as a numerical vector in the form of features for graph nodes, while capturing the semantic and flow aspects of code blocks through graph edge features. Mathematically, this can be defined as

Let $G = (V, E)$ be the constructed CEG, where:

- V represents the set of graph nodes, each corresponding to a code block.
- E denotes the set of graph edges, signifying relationships between code blocks.

For each node $v_i \in V$, we aim to extract feature vectors X_i such that:

$$X_i = \text{FeatureExtractor}(v_i), \forall v_i \in V$$

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.

Additionally, we seek to capture the semantics and flow between code blocks as edge features E_{ij} such that:

$$E_{ij} = \text{EdgeFeatureExtractor}(v_i, v_j), \forall (v_i, v_j) \in E$$

B. Problem 2: Malicious family detection

The second problem entails the classification of the constructed CEGs into malicious or benign families. We employ a GCNN for this classification task. Formally, the problem can be expressed as:

Given a set of CEGs $\{G_1, G_2, \ldots, G_N\}$, each annotated with a class label y_i, where $y_i = 1$ indicates a malicious family and $y_i = 0$ represents a benign family, we aim to learn a classifier f that maps CEGs to class labels:

$$f(G_i) \rightarrow y_i, \forall G_i \in \{G_1, G_2, \ldots, G_N\}$$

The objective is to train a GCNN model to minimize the classification loss:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} (y_i \cdot \log(f(G_i)) + (1 - y_i) \cdot \log(1 - f(G_i)))$$

C. Problem 3: Interpretability via subgraph extraction

The third problem addresses the interpretability of the classification results by generating subgraphs of CEGs that highlight key code blocks responsible for classification decisions. We introduce the GAGE to perform this task. Formally, we aim to extract subgraphs G_s from CEGs G that provide meaningful insights into the classification process:

$$G_s = \text{GAGE}(G, f(G), y_i), \forall G \in \{G_1, G_2, \ldots, G_N\}$$

Where $f(G)$ represents the classification output for CEG G, and y_i is the true class label for G_i.

IV. Model Development

A. CEG construction

The initial phase of our model development involves constructing the Canonical Executable Graph (CEG) through a multi-step process (see Figure 1), which can be divided into two main components: feature extraction for nodes and edge features.

1) Feature extraction for nodes: Following the disassembling of PE files using IDA Pro[3], we extract blocks of assembly instructions within functions. These instruction blocks are then processed using the PalmTree[4] library, a pre-trained model on assembly language that has been trained extensively on CFG and DFG to capture semantic information [25]. The next step involves converting and reducing the dimensionality of these embeddings at the block level. However, directly aggregating or applying weighted sums to instruction-level embeddings generated by PalmTree is unsuitable, as it neglects the sequence's inherent order, potentially resulting in information loss.

[3]https://www.hex-rays.com/products/ida/

[4]https://github.com/palmtreemodel

To address this challenge, we developed AED, a model that combines the strengths of traditional autoencoders and sequence-to-sequence models, incorporating an attention mechanism to adaptively focus on different parts of the input sequence during encoding and decoding. The AED architecture comprises two main components:

- **Attention-Encoder** The encoder employs Convolutional Neural Networks (ConvNets) to capture spatial features within the assembly code sequence. Subsequently, a self-attention mechanism assigns varying weights to sequence segments based on their significance. Mathematically, the encoder's output (E_O) can be expressed as:

$$E_O = \text{Attention}(\text{Conv1D}(B(m, n))) \quad (1)$$

  where $B(m, n)$ represents the embedding of a block generated by PalmTree, where m denotes the number of instructions, and each instruction has an embedding of size n.
- **Decoder** The decoder takes the encoded representation and reconstructs the original input sequence using Conv1DTranspose layers. Similar to the encoder, it employs the attention mechanism to ensure that the generated output focuses on relevant portions of the encoded representation. The decoder's output, which is the reconstruction of instruction embeddings (R_o) from the input (E_o), can be defined as:

$$R_o = \text{Conv1DTranspose}(\text{Attention}(E_O)) \quad (2)$$

After training AED, a feature of the Node can be obtained as:

$$F_{\text{Node}} = AED(B(m, n)) \quad (3)$$

The generated feature vector (F_{Node}) captures the characteristics of corresponding assembly instruction sequences in a lower-dimensional embedding space, retaining essential information such as opcode details, operand types, and operand values. The attention mechanism enables the model to emphasize critical instructions and their relationships, capturing local and global dependencies within the code. Once trained, the AED can obtain embeddings for assembly code sequences.

2) Edge Definitions: Edges within a CEG are pivotal in representing control-flow and data-flow relationships between code blocks, as they are essential for understanding software control flow. We define edges based on the following criteria:

- **Consequent edges (E_C)** Consequent edges link the last block of a function to the first block of the following function, indicating sequential execution.
- **Conditional/fallthrough edges (E_{Cond})** Conditional edges represent control-flow decisions, reflecting branching within code blocks. Specific conditions determine these edges.
- **Intra-function edges (E_{Intra})** Intra-function edges exist within a single function and capture local control flow. They connect blocks based on control dependencies.
- **External edges (E_{External})** External edges connect code blocks across different functions or program units, signifying interactions between them. These edges provide

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.

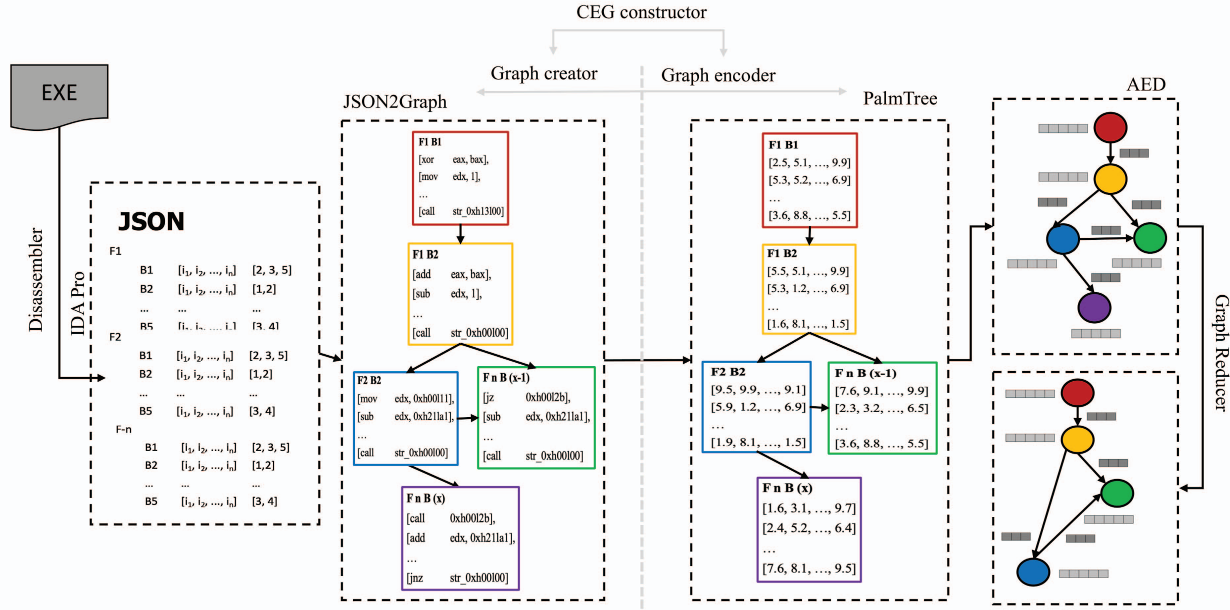

Fig. 1. Pipeline of AED for CEG construction

insights into inter-procedural control flow and data flow, facilitating a deeper understanding of how various parts of the software collaborate or communicate.

Mathematically, the embedding of an edge can be represented as a vector incorporating all the Booleans as mentioned above:

$$F_{\text{Edge}} = [E_C, E_{\text{Cond}}, E_{\text{Intra}}, E_{\text{External}}] \tag{4}$$

CEG construction involves the extraction of node features representing code block characteristics and the definition of edges to model control-flow relationships. This mathematical representation of software structure and behaviour, enhanced by features and edges, enables diverse software analysis tasks, including malware analysis.

B. CEG classification

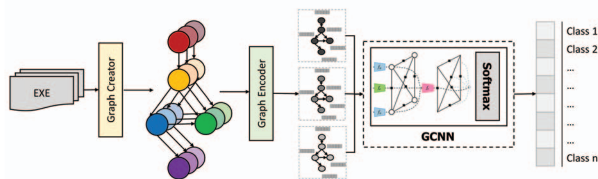

Fig. 2. CEG classification using GCNN

In our research, the classification of CEGs into benign and malicious families is essential. To achieve this, we utilize a GCNN, a robust framework designed for graph-based data classification (Figure 2). The architecture of the GCNN model is summarized in Table I, which provides an overview of its layers, output shapes, and associated parameters.

TABLE I
ARCHITECTURE OF THE GCNN

Layer	Parameters
Input	N/A
Graph convolution	N/A
Graph pooling	N/A
Convolutional layer	Filters = 16, Kernel Size = sum(layer_sizes), Strides = sum(layer_sizes)
Max pooling	Pool Size = 2
Convolutional Layer	Filters = 32, Kernel Size = 5, Strides = 1
Dense	Units = 128, Activation = ”relu”
Dropout	Dropout rate = 0.5
Output	Activation (Softmax)

Following the process outlined in the previous section (Section IV-A) for CEG generation, we employ the GCNN architecture, as depicted in Table I, for graph-based classification. This architecture incorporates graph convolutional layers, convolutional layers, max-pooling operations, and fully connected layers, collectively enabling the model to capture intricate relationships within the graph data. The training of the model is carried out in a batch-based manner, with early stopping mechanisms in place to prevent overfitting. Model performance is rigorously evaluated using an independent test dataset, and critical metrics, including accuracy, loss, precision, recall, and F1-score, are computed for comprehensive assessment.

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.

C. GAGE

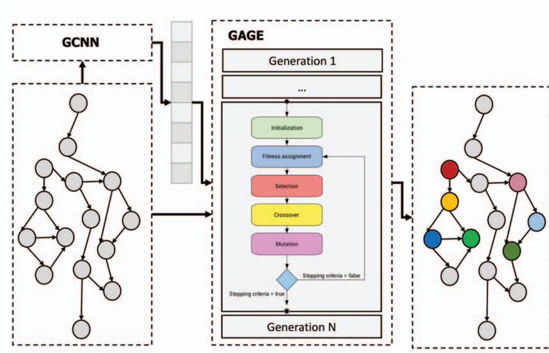

Fig. 3. Subgraph extraction using GA

In this section, we present the genetic algorithm (GA) approach used to enhance graph-based classification through subgraph extraction. The GA iteratively optimizes subgraphs based on a fitness function derived from softmax probabilities obtained during the classification of the original graph. The GA comprises several crucial steps, which we detail mathematically below:

1) Encoding subgraph: Given a parent graph G_p with a set of edges, we represent a subgraph G_s as a chromosome C of length L. Each element C_i within the chromosome corresponds to an edge index from the encoding scheme. The encoding process, including the use of an $EdgeMapping$ to relate edge indices to actual edges, is defined as:

$$C = [C_1, C_2, \ldots, C_L] \tag{5}$$

where C_i is an integer representing an edge index, and $EdgeMapping(C_i)$ maps it to an edge in G_p.

2) Crossover: Crossover is a genetic operator that combines the chromosomes of two-parent subgraphs, C_a and C_b, to generate a child chromosome C_c. The crossover operation can be formulated as follows:

$$C_c = \text{Crossover}(C_a, C_b) \tag{6}$$

3) Mutation: Mutation introduces diversity into the population by randomly altering specific elements within a chromosome. The mutated chromosome C_m is expressed as:

$$C_m = \text{Mutation}(C, \text{mutation_rate}) \tag{7}$$

4) Decoding: The decoding process constructs the decoded subgraph G_d from the chromosome C using an encoding-decoding mapping function. This mapping translates edge indices back to their respective edges in the parent graph G_p using the $EdgeMapping$:

$$G_d = \text{Decode}(C, \text{EdgeMapping}) \tag{8}$$

5) Fitness Calculation: Fitness evaluation measures the quality of the decoded subgraph G_d concerning its classification performance. The fitness function is defined as the Euclidean distance between the softmax probabilities of G_d and G_p, calculated across all classes:

$$\text{Fitness}(G_d, G_p) = \sqrt{\sum_{i=1}^{N} (P_i - C_i)^2} \tag{9}$$

where N represents the number of classes, P_i is the softmax probability of class i for G_p, and C_i is the softmax probability of class i for G_d.

6) Selection of Fittest Individuals: The selection process identifies the fittest subgraphs within the population based on their computed fitness values. The top N_{top} subgraphs, corresponding to the lowest fitness values, are chosen to proceed to the next generation.

7) Creating a New Population: To evolve the population towards subgraphs with improved classification performance, crossover, mutation, and selection are applied iteratively. This iterative process continues for a specified number of generations, resulting in the generation of increasingly optimized subgraphs.

V. Result and Discussion

A. Dataset

The dataset for this research comprises malicious executables obtained from MalShare[5] and VirusShare[6]. It comprises 612 benign files (13.6GB) and 1,799 malicious files (15.1 GB) attributed to the Bladabindi, Bundlore, Downloadadmin, Emotet, Gamarue, and Firseria malware families. We utilized IDA Pro, a commercial disassembler, to disassemble our compiled executables and obtain the corresponding assembly functions.

To train our AED, we employed a dataset consisting of 0.8 million assembly code blocks, with each block limited to a maximum of 512 instructions. On average, each CEG comprises 546 nodes and 3,567 edges. For our model evaluation, we divided the dataset into an 80-20% train-test split. Subsequently, the training set was further split into an 80-20% training-validation split for model development and validation.

B. Discriminative power analysis

In this section, we present a comparative analysis of the performance of our proposed model GAGE against the state-of-the-art CFGExplainer. We evaluate their discriminative power using precision (P), recall (R), and F1-Score $(F1)$ metrics for various malware families. The results are summarized in Table II.

We begin by examining the classification performance for each malware family individually, using the following formulas:

[5]https://malshare.com
[6]https://virusshare.com

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.

TABLE II
DISCRIMINATIVE POWER METRICS

Malware-Family	Algorithm	Precision	Recall	F1-Score
Gamarue	CFGExplainer	0.46	0.25	0.32
	GAGE	0.68	0.44	0.53
Firseria	CFGExplainer	0.93	0.98	0.95
	GAGE	0.98	0.98	0.98
Bundlore	CFGExplainer	1.00	0.94	0.97
	GAGE	1.00	0.96	0.98
Emotet	CFGExplainer	0.95	0.89	0.92
	GAGE	0.89	0.86	0.88
Benign	CFGExplainer	0.69	0.84	0.76
	GAGE	0.75	0.89	0.81
Downloadadmin	CFGExplainer	0.93	0.98	0.96
	GAGE	0.96	0.99	0.97
Bladabindi	CFGExplainer	0.72	0.60	0.65
	GAGE	1.00	0.83	0.91
Average	**CFGExplainer**	**0.81**	**0.78**	**0.79**
	GAGE	**0.90**	**0.85**	**0.87**
Accuracy	**CFGExplainer**	**0.83**		
	GAGE	**0.87**		

$$Precision = \frac{TP}{TP + FP} \tag{10}$$

$$Recall = \frac{TP}{TP + FN} \tag{11}$$

$$F1 - Score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{12}$$

Where:

- TP is the number of true positives.
- FP is the number of false positives.
- FN is the number of false negatives.

We found that GAGE outperforms CFGExplainer for almost every malware family regarding precision, recall, and F1-Score (see Table II). GAGE's performance on Emotet may differ due to CEGs' broader syntactical and semantical scope compared to the execution-path-centric approach of CFGs used in CFGExplainer, highlighting the need for further research on malware-graph interactions for different malware families.

To provide an overall assessment, we calculate the average precision, recall, and F1-Score across all malware families. Here are the formulas:

$$AveragePrecision = \frac{1}{N}\sum_{i=1}^{N} Precision_i \tag{13}$$

$$AverageRecall = \frac{1}{N}\sum_{i=1}^{N} Recall_i \tag{14}$$

$$AverageF1 - Score = \frac{1}{N}\sum_{i=1}^{N} F1 - Score_i \tag{15}$$

Where:

- N is the number of malware families.
- $Precision_i$, $Recall_i$, and $F1 - Score_i$ are the precision, recall, and F1-Score values for the i-th malware family.

Here, GAGE consistently demonstrates superior performance compared to CFGExplainer, with higher values for precision, recall, and F1-Score. In terms of accuracy, which represents the overall classification correctness, GAGE achieves a higher accuracy score compared to CFGExplainer.

The results of our performance evaluation indicate that GAGE outperforms CFGExplainer across multiple malware families, achieving higher precision, recall, F1-Score, and accuracy. These findings underscore the effectiveness of our proposed model in the context of malware classification. GAGE's superior discriminative power makes it a valuable tool for identifying and classifying various malware families, providing enhanced security in the face of evolving threats.

C. Robustness assessment

TABLE III
EXTRACTED SUBGRAPH FEATURE TO CALCULATE ROBUSTNESS SCORE.

Feature	Description
Average node features	The average of node features
Edge Count	The total number of edges in the graph.
Self-Loop Count	The number of self-loops in the graph.
Minimum Degree	The minimum degree of nodes in the graph.
Minimum In-Degree	The minimum in-degree of nodes in the graph.
Minimum Out-Degree	The minimum out-degree of nodes in the graph.
Average Degree	The average degree of nodes in the graph.
Average In-Degree	The average in-degree of nodes in the graph.
Average Out-Degree	The average out-degree of nodes in the graph.
Maximum Degree	The maximum degree of nodes in the graph.
Maximum In-Degree	The maximum in-degree of nodes in the graph.
Maximum Out-Degree	The maximum out-degree of nodes in the graph.

To assess the robustness of the explanations generated in subgraphs, we extracted various features from these subgraphs, as defined in Table III. Subsequently, we computed the Minimum Mean Discrepancy (MMD) score as a measure of robustness. The MMD between two sets of data can be calculated using the following formula:

$$MMD(X, Y) = \left\| \frac{1}{n_X}\sum_{i=1}^{n_X} \phi(x_i) - \frac{1}{n_Y}\sum_{j=1}^{n_Y} \phi(y_j) \right\|_2^2 \tag{16}$$

In this formula:

- X and Y are the data points we want to compare.
- n_X and n_Y are the number of data points in sets X and Y, respectively.
- x_i and y_j are individual data points in sets X and Y.
- $\phi(\cdot)$ is a feature map that maps data points into a higher-dimensional space.
- $\|\cdot\|_2$ denotes the Euclidean norm (L2 norm), the square root of the sum of squared values.

The MMD measures the difference between the feature distributions of the two datasets X and Y. It quantifies how well the data points from X and Y are separated in the feature space defined by $\phi(\cdot)$. The smaller the MMD value, the more similar the distributions of X and Y are in the feature space.

In Figure 4, we present the robustness scores compared across different malware families for varying numbers of

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
ROBUSTNESS SCORES ACROSS CLASSES AND COMPARISON BETWEEN CFGEXPLAINER AND GAGE USING VARYING DATA SIZES (1 TO 5 SUBGRAPHS)

Class 1	Class 2	Model	#1	#2	#3	#4	#5	Average
Benign	Bladabindi	CFGExplainer	1.5543	0.7369	0.3386	0.3330	0.3330	0.6591
		GAGE	1.9994	0.6033	0.4411	0.2763	0.2763	0.7192
Benign	Bundlore	CFGExplainer	1.2645	0.5018	0.2267	0.2567	0.2567	0.5012
		GAGE	1.5844	1.1256	0.5205	0.3411	0.3411	0.7825
Benign	Downloadadmin	CFGExplainer	1.2816	0.5052	0.3944	0.2092	0.2092	0.5199
		GAGE	1.7533	0.8976	0.3156	0.3424	0.3424	0.7302
Benign	Emotet	CFGExplainer	1.8396	0.7594	0.2701	0.3300	0.3300	0.7058
		GAGE	1.8969	0.8744	0.5971	0.4938	0.4938	0.8712
Benign	Firseria	CFGExplainer	1.7296	0.4858	0.1948	0.1239	0.1239	0.5316
		GAGE	1.9665	1.0273	0.6955	0.6822	0.6822	1.0107
Benign	Gamarue	CFGExplainer	1.7305	0.5022	0.3511	0.5241	0.5241	0.7264
		GAGE	1.9470	0.9196	0.6569	0.5819	0.5819	0.9374
Bladabindi	Bundlore	CFGExplainer	1.8360	0.4603	0.2071	0.1261	0.1261	0.5511
		GAGE	1.9999	0.5140	0.2462	0.3097	0.3097	0.6759
Bladabindi	Downloadadmin	CFGExplainer	1.8382	0.4594	0.6298	0.3204	0.3204	0.7136
		GAGE	1.9999	0.6702	0.5973	0.6438	0.6438	0.9110
Bladabindi	Emotet	CFGExplainer	1.2777	0.3283	0.4564	0.3322	0.3322	0.5453
		GAGE	1.9998	1.0183	0.6879	0.6539	0.6539	1.0027
Bladabindi	Firseria	CFGExplainer	0.7900	0.7978	0.7438	0.2661	0.2661	0.5727
		GAGE	1.9677	1.0948	0.9265	0.9453	0.9453	1.1759
Bladabindi	Gamarue	CFGExplainer	0.7897	0.8955	0.7546	0.6432	0.6432	0.7452
		GAGE	1.9997	0.9440	0.8394	0.8268	0.8268	1.0873
Bundlore	Downloadadmin	CFGExplainer	0.0474	0.0134	0.2293	0.2275	0.2275	0.1490
		GAGE	1.0054	0.6276	0.6003	0.5814	0.5814	0.6792
Bundlore	Emotet	CFGExplainer	1.5655	0.4064	0.3020	0.4533	0.4533	0.6361
		GAGE	1.9783	1.3101	0.8047	0.5597	0.5597	1.0425
Bundlore	Firseria	CFGExplainer	1.9635	0.6627	0.4492	0.2553	0.2553	0.7172
		GAGE	1.9996	1.3952	1.0323	0.8830	0.8830	1.2386
Bundlore	Gamarue	CFGExplainer	1.9635	0.7297	0.6004	0.6913	0.6913	0.9352
		GAGE	1.7730	1.2301	0.8073	0.5595	0.5595	0.9858
Downloadadmin	Emotet	CFGExplainer	1.5591	0.3978	0.3856	0.3957	0.3957	0.6267
		GAGE	1.9933	1.0467	0.6443	0.5345	0.5345	0.9506
Downloadadmin	Firseria	CFGExplainer	1.9642	0.6613	0.4227	0.1993	0.1993	0.6893
		GAGE	1.9999	1.1094	0.7824	0.6798	0.6798	1.0502
Downloadadmin	Gamarue	CFGExplainer	1.9640	0.7309	0.5814	0.3988	0.3988	0.8147
		GAGE	1.9856	0.9951	0.6359	0.5105	0.5105	0.9275
Emotet	Firseria	CFGExplainer	1.8384	0.6343	0.3215	0.2616	0.2616	0.6634
		GAGE	1.6169	0.5967	0.5186	0.5031	0.5031	0.7476
Firseria	Gamarue	CFGExplainer	0.0177	0.5050	0.3986	0.4432	0.4432	0.3615
		GAGE	1.9997	1.0946	0.7195	0.6155	0.6155	1.0089
	Average	CFGExplainer						0.6182
		GAGE						**0.9267**

extracted subgraphs. Our findings indicate that the proposed algorithm consistently achieves high robustness scores across most scenarios, with a few exceptions. Similarly, we conducted a comparative analysis among different malware families and observed comparatively better robustness scores for the proposed algorithm (refer to Figure 5).

Table IV displays the robustness scores between all benign and malware families across different data sizes. We also calculate the average for each combination and the final average to facilitate a direct comparison between CFGExplainer and GAGE. Our results show that CFGExplainer achieves a 61.82% robustness score, while GAGE attains an impressive 92.67% signifying its superiority.

D. Interpretability analysis

In this section, we analyze the explainability offered by the proposed model from various perspectives (refer to Table V for acronyms).

TABLE V
LIST OF ACRONYMS RELATED TO CODE ANALYSIS AND THEIR DEFINITIONS

Acronym	Definition
XOR	Exclusive OR (a logical operation)
ROL	Rotate Left (an assembly instruction)
ROR	Rotate Right (an assembly instruction)
MOV	Move (an assembly instruction)
LEA	Load Effective Address (an assembly instruction)
TEST	Test (an assembly instruction)
CMP	Compare (an assembly instruction)
JNZ	Jump if Not Zero (an assembly instruction)
JZ	Jump if Zero (an assembly instruction)
JB	Jump if Below (an assembly instruction)
MUTAG	A dataset name related to mutagenic compounds

1) Code obfuscation: Malware frequently utilizes code obfuscation techniques to obstruct static analysis and elude detection mechanisms. A prominent instance from the extracted code blocks involves the application of XOR operations,

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.

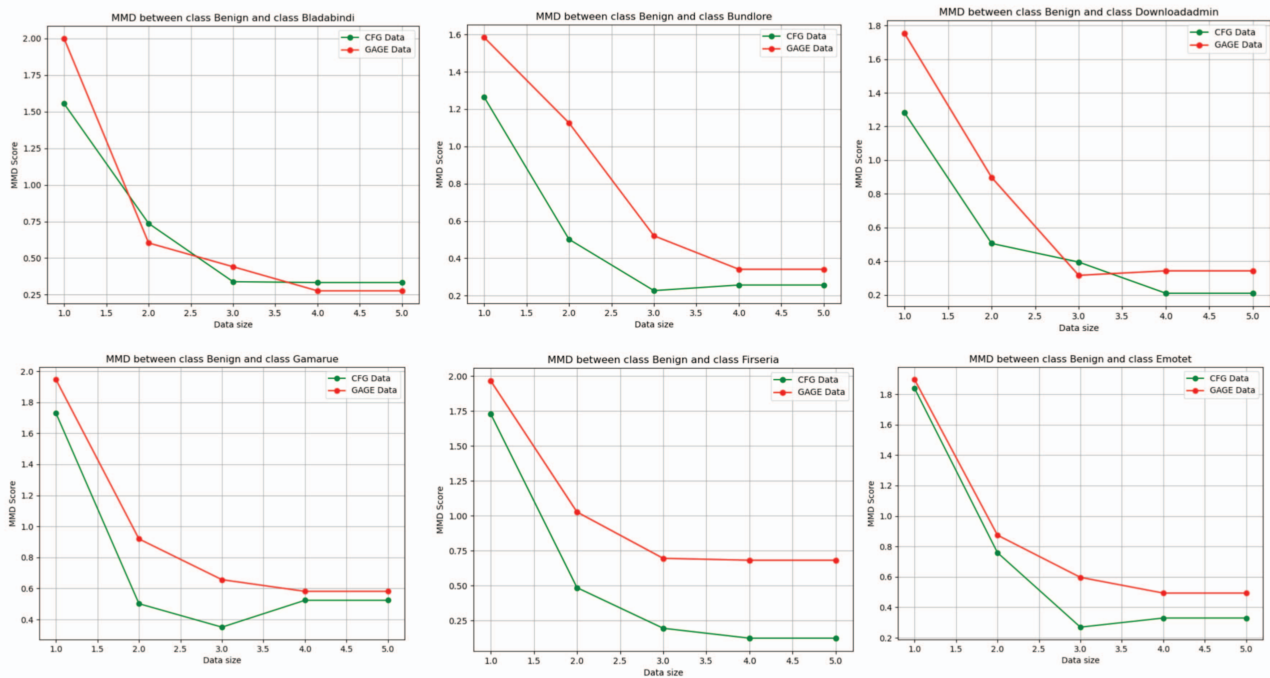

Fig. 4. Robustness score/MMD between benign and various malware families.

which are commonly used for straightforward data encoding and decoding. Additionally, the employment of arithmetic and logic instructions, such as ROL and ROR, particularly within loops, is discernible in the extracted code, potentially signalling a decoding routine. Specific obfuscation instructions have been observed in several examples from the Firseria, Emotet, and DownloadAdmin families, as illustrated in Figure 6.

2) Evasion techniques: The proposed model, GAGE, identifies blocks that unveil evasion tactics, notably the employment of jump instructions to formulate a complex CFG, thereby complicating static analysis. For example, dynamic jumps and potentially packed or encrypted payloads, exemplified by $jmp : ds : __imp_DllFunctionCall$ in the Gamarue family, are deemed suspicious as they are frequently utilized to circumvent detection and analysis. Such instructions suggest the executable's use of external libraries or functions, potentially engaging with system-level functionalities or interacting with other processes.

3) Data manipulation: Data and memory management play a crucial role in the functioning of malware. A prevalent utilization of MOV and LEA instructions was noted, which may involve transferring malicious payloads or altering memory addresses. Moreover, employing $TEST$, CMP, and conditional jump instructions, such as JNZ, JZ, and JB, could establish conditional logic derived from the manipulated data. Notably, in the extracted code from the Gamarue family, an extensive use of MOV commands was observed (Figure 6).

4) Unpacking, shellcode, or payload execution: Recognizing patterns that suggest shellcode execution or the unpacking of additional payloads is vital. This may encompass a blend of memory operations, function calls, and jumps that execute data in memory. For example, the utilization of hardcoded values, often in hexadecimal, might be linked with specific operations, and magic numbers are atypical in benign applications. Such signs were observed in the Gamarue family samples (Figure 6). The Firseria samples display an unusual quantity of calls, jumps, and conditional checks. While stack operations are common, they can also be employed in shellcode or to set up function calls with particular arguments.

5) Comparison with benign samples: Benign samples, scrutinized through the extraction of subgraphs via the GAGE algorithm, display markedly distinct attributes in comparison to malicious samples (Figure 9). The code blocks within benign samples are systematically structured and organized, executing particular operations or tasks (Figure 8). The following are some pivotal findings from the malicious code extracted by GAGE:

- **Code architecture:** Benign samples generally display a modular and systematic code structure engineered to execute specific functionalities, which stands in stark contrast to the frequently obfuscated or packed code observed in malware.
- **Handling exceptions:** Instructions pertinent to exception handling, such as $pushoffset__except_handler4$, are common in benign samples, ensuring the proper management of runtime errors and exceptions.

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.

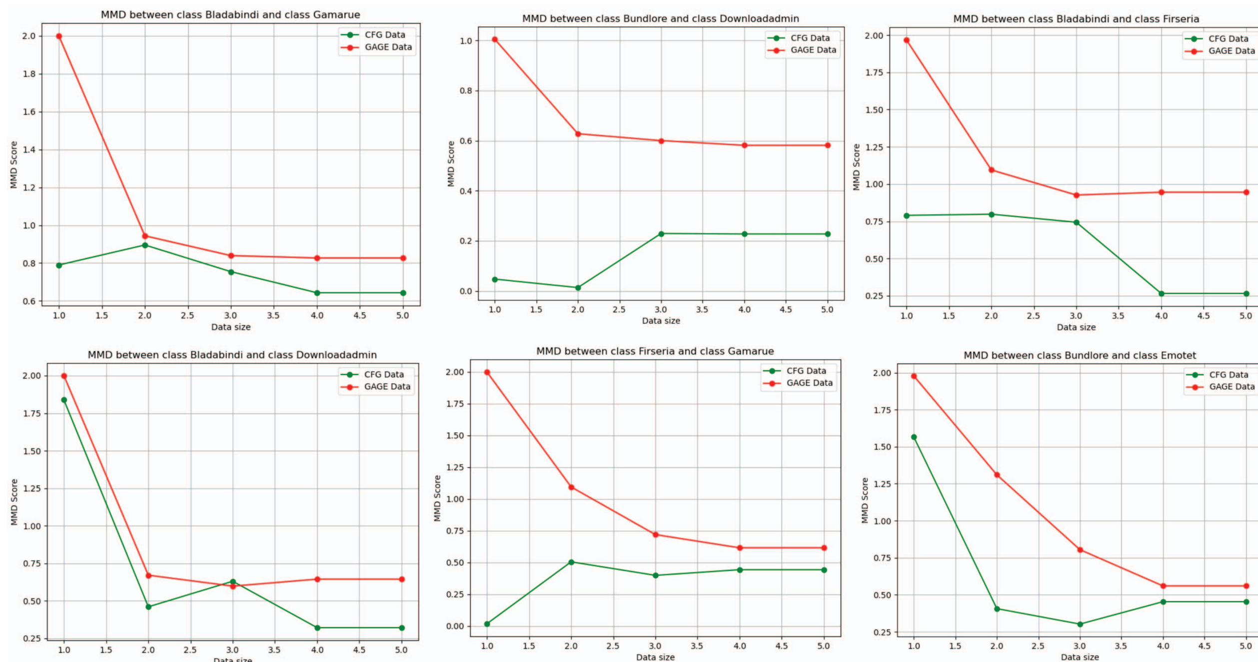

Fig. 5. Robustness score/MMD between two different malware families

```
"mov esi [ebp+arg_4]",
"mov edi [ebp+arg_0]",
"mov [esp+54h+var_20] 7D820636h",
"mov ebx [esp+54h+var_1C]",
"mov [esp+54h+var_28] eax",
"mov eax [esp+54h+var_18]",
"mov [esp+54h+var_2C] eax",
"mov eax 7D820636h",
"mov [esp+54h+var_18] 0",
"mov [esp+54h+var_1C] 3076148Dh",
"add eax 827DF9CAh",
"cmp ecx eax",
"mov [esp+54h+var_30] ebx",
"mov [esp+54h+var_34] ecx",
"mov [esp+54h+var_38] edx",
"mov [esp+54h+var_3C] esi",
"mov [esp+54h+var_40] edi",
"jnz loc_401097"
],
```

```
"4198896_0": [
    "jmp ds:__imp_DllFunctionCall"
],
"4576352_129": [
    "call ds:__vbaErrorOverflow"
]
"4576352_11": [
    "push offset dword_460010",
    "push offset dword_40172C",
    "call ds:__vbaNew2"
],
"4576352_16": [
    "mov edx [ebp+var_D0]",
    "push offset aFavoritesFolde",
    "lea ecx [ebp+var_4C]",
    "call esi",
    "push eax",
    "call edi",
```

```
"4216868_37": [
    "xor eax eax",
    "lea ecx [esp+0DCh+var_64]",
    "push 0",
    "inc eax",
    "push eax",
    "call sub_4016CA",
    "xor eax eax",
    "lea ecx [esp+0DCh+var_4C]",
    "push 0",
    "inc eax",
    "push eax",
    "call sub_4016CA",
    "xor eax eax",
    "lea ecx [esp+0DCh+Dst]",
    "push 0",
    "inc eax",
```

```
    "add esi 0FF4A7FB9h",
    "and esi 448960Bh",
    "add edx 4682465Dh",
    "mov [esp+3Ch+var_20] edx",
    "xor esi 6E6D5C0h",
    "mov [esp+3Ch+var_30] eax",
    "mov [esp+3Ch+var_34] ecx",
    "mov [esp+3Ch+var_38] esi",
    "jnz loc_402335"
],
"4204928_4": [
    "mov eax 1E15A90Eh",
    "mov ecx 115F53Dh",
    "mov edx [esp+54h+var_3C]",
    "sub ecx edx",
    "mov esi [esp+54h+var_24]",
    "add esi 43FFB12Ch",
```

Fig. 6. Malware families with malicious code interpretability. Pink lines show extensive use of MOV commands, red shows dynamic calls, blue shows magic numbers used in malicious code, and yellow shows XOR obfuscation technique by malicious files. These samples are from malware families (Gamaru and Firseria).

- **Security protocols:** Instructions concerning security, such as $moveax, ___security_cookie$, along with subsequent operations, manage security cookies, a strategy employed in benign software to thwart buffer overflow attacks (Figure 8).
- **Memory administration:** Efficient memory management is demonstrated through instructions that manage local variables and function calls, a characteristic typically observed in benign software, e.g., managing stack pointer (check sky-blue lines in Figure 8).

E. Validation using MUTAG dataset

Without ground truth for evaluating the interpretability on malicious file datasets, we turn to real-world data, specifically the MUTAG dataset [26], to validate our results. The MUTAG dataset comprises a collection of nitroaromatic compounds designed for graph classification to distinguish between mutagenic and non-mutagenic compounds. Our objective is to assess the interpretability by identifying subgraphs or nodes corresponding to mutagenic behaviour in graph structures.

We initiate this process by performing graph classification, achieving favourable discriminative power. Subsequently, we employ our proposed algorithm, GAGE, to obtain the interpretability.

After training the model for classification and extracting subgraphs for both mutagenic and non-mutagenic classes, we obtain meaningful results. Non-mutagenic compounds within the MUTAG dataset are primarily composed of carbon (C), nitrogen (N), oxygen (O), and hydrogen (H) atoms [27], [28]. These elements are commonly found in various organic compounds and the building blocks for numerous biological molecules. GAGE effectively highlights C and O nodes in the

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.

Fig. 7. Malware families with malicious subgraph interpretability. Red nodes and edges represent the most suspicious code blocks in their respective executables of the Emotet, Firseria, Downloadadmin, and Gamarue malware families (from left to right).

Fig. 8. Interpretability of benign sample. Green nodes indicate code-blocks highlighted by GAGE.

```
"268469195_0": [
    "push offset __except_handler4",
    "push large dword ptr fs:0",
    "mov eax [esp+8+arg_4]",
    "mov [esp+8+arg_4] ebp",
    "lea ebp [esp+8+arg_4]",
    "sub esp eax",
    "push ebx",
    "push esi",
    "push edi",
    "mov eax ___security_cookie",
```

Fig. 9. Interpretability of extracted code from a benign sample. The green line relates to exception handling code, sky-blue pertains to stack pointer management, and the blue line illustrates security-related checkpoints. In benign samples, code blocks highlighted by GAGE indicate these aspects.

case of non-mutagenic compounds (Figure 10).

In contrast, mutagenic compounds within the MUTAG dataset exhibit a broader spectrum of atoms than non-mutagenic ones. While carbon (C), nitrogen (N), oxygen (O), and hydrogen (H) atoms remain prevalent, mutagenic compounds can also incorporate halogens such as fluorine (F), chlorine (Cl), bromine (Br), and iodine (I) [27], [28].

Fig. 10. Non-mutagenic compound: Blue nodes represent typical nodes, while red nodes, highlighted by GAGE, correspond to carbon (C) and oxygen (O) atoms.

Fig. 11. Mutagenic compound: Blue nodes represent typical nodes, while red nodes, highlighted by GAGE, correspond to halogen atoms.

In mutagenic cases, GAGE successfully identifies Cl and H atoms (refer to Figure 11).

F. Model comparison with other tools

Table VI presents a comparative analysis of our model with other benchmark malware detection models. Models not incorporating explanations, [29], [30] demonstrate high evaluation metrics. However, further analysis is necessary to understand the rationale behind malware identification and determine the malware family. Model [31] offers explanations based on system calls and exhibits relatively high accuracy, although its precision is comparatively low. This indicates a propensity to overpredict the positive class, resulting in numerous false alarms or incorrect identifications. While the model's overall accuracy appears high, its precision suggests a need for improvement in identifying true positive cases without generating excessive false positives. Models not utilizing graphs, such as

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.

TABLE VI
GAGE'S RESULTS COMPARISON WITH OTHER BENCHMARK TOOLS E.G., CONVOLUTIONAL LONG SHORT-TERM MEMORY NETWORK (CONV-LSTM), GRADIENT-WEIGHTED CLASS ACTIVATION MAPPING (GRAD-CAM), MALCONV (CONVOLUTIONAL NEURAL NETWORK FOR MALWARE)

Ref	Input data	Model/ Algorithm	Precision	Recall	F1-Score	Accuracy	Explainability method	Explainability evaluation
[32]	Gray scale images	CNN	72.6	71.5	72.0	71.8	No	No
[33]	Malware images	Grad-CAM	94.7	94.3	94.5	94.4	Most influencing pixels, heatmap	Yes
[29]	Byte sequences	CNN	95.9	96.3	96.1	96.1	No	No
[30]	Byte sequences	CNN	93.2	93.2	93.2	93.2	No	No
[34]	Malware images	MalConv	87.1	–	87.3	–	Heatmap	No
[31]	System calls	ANN	85.0	96.0	–	94.0	Most influencing system call's tags	Yes
[35]	Features series	Conv-LSTM	93.8	51.4	67.9	89.2	Subgraph	No
[36]	CFG	GNN	–	–	92.7	89.6	Subgraph	Yes
GAGE	CEG	GCNN	90.0	85.0	87.0	87.0	Subgraph	Yes

the malware image based model [33], achieve relatively better accuracy owing to their robust CNN architectures. However, their explainability, which involves highlighting influential pixels in images, only provides meaningful insights to malware analysts if they reconnect those pixels to the corresponding PE files. This process is time-consuming and requires deep knowledge of the developed model.

Models employing graphs and providing explanations in terms of subgraphs offer convenience to malware analysts by directly extracting sequences of malicious code. Unlike other models, they obviate the need to explore the model further. However, only a few models consider malicious graphs as input, as processing malicious graphs necessitates a different approach due to their mixture of benign and malicious code within a single PE. Besides CFGExplainer, models [35] utilize graph inputs and provide explanations as subgraphs, claiming better accuracy. However, the model [35] does not evaluate the quality of the generated explanations. While its precision, accuracy, and F1-score suggest overall good performance, its low recall underscores a need to improve capturing more true positive cases. Model [36], although evaluating their explanation, is only applicable for Android-based malware and not PE.

The proposed model, GAGE, demonstrates that including explainability features does not significantly compromise its accuracy. The precision, recall, and F1-score metrics demonstrate GAGE's ability to maintain high discriminative power while providing interpretable explanations. GAGE's ability to offer explanations in terms of subgraphs could offer significant advantages in understanding the underlying reasons for malware classifications. Therefore, the explainability provided by GAGE is worth the slight decrease in accuracy, as it enhances the transparency and trustworthiness of the model's outputs, facilitating informed decision-making by malware analysts and cybersecurity professionals.

VI. CONCLUSION

Our proposed algorithm is specifically designed to address the unique characteristics of malicious files, such as the non-applicability of gradient and perturbation algorithms, surrogation, and division methods. The proposed algorithm captures syntax and semantic-level knowledge through node encoding, and the resulting graph structure, CEG, retains all the details of the executable code. Beyond malware detection, our algorithm significantly contributes to fields where graphs represent a pivotal yet complex data structure. It effectively addresses issues such as graph segregation and gradient problems associated with previous XAI methods, offering broader applicability and enhancing the interpretability of graph-based learning and explanation approaches across diverse domains.

The proposed algorithm has achieved superior discriminative power with an 87% accuracy rate and a lower false positive rate. Furthermore, GAGE provides interpretability, yielding a robustness score of 97.67%, a crucial aspect for distinguishing between different malware families. We conducted a manual analysis of the code extracted by the proposed model and found it highly valuable for reverse engineering purposes. The extracted subgraph contains some unfamiliar and suspicious elements, which can be essential for further investigation. In addition, we also applied the proposed algorithm to a real-world dataset, MUTAG, and obtained meaningful results in terms of interpretability.

In the future, we will focus on developing a graph taxonomy to differentiate between benign and various malware families. Taxonomies such as bunch, ring, and Barabasi-Albert (BA), when combined, can reveal obfuscation and encoding schemes, providing insights into the executable's origin and association with different malware families.

REFERENCES

[1] D. Ucci, L. Aniello, and R. Baldoni, "Survey of machine learning techniques for malware analysis," *Computers & Security*, vol. 81, pp. 123–147, 2019.

[2] S. Cesare and Y. Xiang, "Classification of malware using structured control flow," in *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107*. Citeseer, 2010, pp. 61–70.

[3] R. Sihwail, K. Omar, and K. Z. Ariffin, "A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis," *Int. J. Adv. Sci. Eng. Inf. Technol*, vol. 8, no. 4-2, pp. 1662–1671, 2018.

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.

[4] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—a state of the art survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–48, 2019.

[5] A. Shalaginov, S. Banin, A. Dehghantanha, and K. Franke, "Machine learning aided static malware analysis: A survey and tutorial," *Cyber threat intelligence*, pp. 7–45, 2018.

[6] M. Ijaz, M. H. Durad, and M. Ismail, "Static and dynamic malware analysis using machine learning," in *2019 16th International bhurban conference on applied sciences and technology (IBCAST)*. IEEE, 2019, pp. 687–691.

[7] Z. Zhang, P. Qi, and W. Wang, "Dynamic malware analysis with feature engineering and feature learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 01, 2020, pp. 1210–1217.

[8] D. Castelvecchi, "Can we open the black box of ai?" *Nature News*, vol. 538, no. 7623, p. 20, 2016.

[9] K. D. Cooper, T. J. Harvey, and T. Waterman, "Building a control-flow graph from scheduled assembly code," Tech. Rep., 2002.

[10] D. C. Zaretsky, G. Mittal, R. Dick, and P. Banerjee, "Generation of control and data flow graphs from scheduled and pipelined assembly code," in *Languages and Compilers for Parallel Computing: 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005, Revised Selected Papers 18*. Springer, 2006, pp. 76–90.

[11] J. Yan, G. Yan, and D. Jin, "Classifying malware represented as control flow graphs using deep graph convolutional neural network," in *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2019, pp. 52–63.

[12] M. H. Nguyen, D. Le Nguyen, X. M. Nguyen, and T. T. Quan, "Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning," *Computers & Security*, vol. 76, pp. 128–155, 2018.

[13] J. D. Herath, P. P. Wakodikar, P. Yang, and G. Yan, "Cfgexplainer: Explaining graph neural network-based malware classification from control flow graphs," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022, pp. 172–184.

[14] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in computer virology*, vol. 7, no. 4, pp. 233–245, 2011.

[15] M. Hassen and P. K. Chan, "Scalable function call graph-based malware classification," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 239–248.

[16] T. Wüchner, M. Ochoa, and A. Pretschner, "Malware detection with quantitative data flow graphs," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 271–282.

[17] ——, "Robust and effective malware detection through quantitative data flow graph metrics," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings 12*. Springer, 2015, pp. 98–118.

[18] F. Baldassarre and H. Azizpour, "Explainability techniques for graph convolutional networks," *arXiv preprint arXiv:1905.13686*, 2019.

[19] P. E. Pope, S. Kolouri, M. Rostami, C. E. Martin, and H. Hoffmann, "Explainability methods for graph convolutional neural networks," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 10 772–10 781.

[20] Q. Huang, M. Yamada, Y. Tian, D. Singh, and Y. Chang, "Graphlime: Local interpretable model explanations for graph neural networks," *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[21] Y. Zhang, D. Defazio, and A. Ramesh, "Relex: A model-agnostic relational model explainer," in *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society*, 2021, pp. 1042–1049.

[22] M. Vu and M. T. Thai, "Pgm-explainer: Probabilistic graphical model explanations for graph neural networks," *Advances in neural information processing systems*, vol. 33, pp. 12 225–12 235, 2020.

[23] T. Schnake, O. Eberle, J. Lederer, S. Nakajima, K. T. Schütt, K.-R. Müller, and G. Montavon, "Higher-order explanations of graph neural networks via relevant walks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 11, pp. 7581–7596, 2021.

[24] H. Yuan, J. Tang, X. Hu, and S. Ji, "Xgnn: Towards model-level explanations of graph neural networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 430–438.

[25] X. Li, Y. Qu, and H. Yin, "Palmtree: learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3236–3251.

[26] A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch, "Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity," *Journal of Medicinal Chemistry*, vol. 34, no. 2, pp. 786–797, 1991. [Online]. Available: https://doi.org/10.1021/jm00106a046

[27] R. T. LaLonde, L. Bu, A. Henwood, J. Fiumano, and L. Zhang, "Bromine-, chlorine-, and mixed halogen-substituted 4-methyl-2 (5 h)-furanones: Synthesis and mutagenic effects of halogen and hydroxyl group replacements," *Chemical research in toxicology*, vol. 10, no. 12, pp. 1427–1436, 1997.

[28] S. Stolzenberg and C. Hine, "Mutagenicity of 2-and 3-carbon halogenated compounds in the salmonella/mammalian-microsome test," *Environmental Mutagenesis*, vol. 2, no. 1, pp. 59–66, 1980.

[29] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware detection by eating a whole exe. arxiv," *arXiv preprint arXiv:1710.09435*, 2017.

[30] M. Krčál, O. Švec, M. Bálek, and O. Jašek, "Deep convolutional malware classifiers can learn from raw executables and labels only," 2018.

[31] L. Pirch, A. Warnecke, C. Wressnegger, and K. Rieck, "Tagvet: Vetting malware tags using explainable machine learning," in *Proceedings of the 14th European Workshop on Systems Security*, 2021, pp. 34–40.

[32] Y. Mourtaji, M. Bouhorma, and D. Alghazzawi, "Intelligent framework for malware detection with convolutional neural network," in *Proceedings of the 2nd International Conference on Networking, Information Systems & Security*, 2019, pp. 1–6.

[33] G. Iadarola, R. Casolare, F. Martinelli, F. Mercaldo, C. Peluso, and A. Santone, "A semi-automated explainability-driven approach for malware analysis through deep learning," in *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2021, pp. 1–8.

[34] S. Bose, T. Barao, and X. Liu, "Explaining ai for malware detection: Analysis of mechanisms of malconv," in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–8.

[35] I. A. Khan, N. Moustafa, D. Pi, K. M. Sallam, A. Y. Zomaya, and B. Li, "A new explainable deep learning framework for cyber threat discovery in industrial iot networks," *IEEE Internet of Things Journal*, 2021.

[36] J. Fairbanks, A. Orbe, C. Patterson, J. Layne, E. Serra, and M. Scheepers, "Identifying att&ck tactics in android malware control flow graph through graph representation learning and interpretability," in *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 2021, pp. 5602–5608.

Authorized licensed use limited to: McGill Libraries. Downloaded on October 24,2024 at 10:46:48 UTC from IEEE Xplore. Restrictions apply.